Massachusetts Institute of Technology
Artificial Intelligence Laboratory

Memo No. 456

January 1978

FLOYD-HOARE VERIFIERS 'CONSIDERED HARMFUL'

by Howard E. Shrobe

Abstract:

The Floyd-Hoare methodology completely dominates the field of program verification and has contribute much to our understanding of how programs might be analyzed. Useful but limited verifiers have been developed using Floyd-Hoare techniques. However, it has long been known that it is difficult to handle side effects on shared data structures within the Floyd-Hoare framework. Most examples of successful Floyd-Hoare verifications have avoided such situations. A recent thesis by Suzuki attempted to state the Floyd-Hoare axioms for assignment to complex data structures, similar statements have been used by London. This paper demonstrates an error in these formalizations and suggests a different style of verification.

Recently Floyd-Hoare logic has been used as the philosophical underpinning for language design efforts. Some designers suggest that one can measure a language's perspicuity by the simplicity of its Floyd-Hoare axioms. Unfortunately, these researchers are considering a very narrow interpretation of the the Floyd-Hoare methodology based on the philosophy that the effect of a program statement can be determined by local syntactic inspection. We show that there is a trade-off between such syntactitc locality and abstraction, forcing language designers to chose between the narrow verification framework and the ability to capture abstraction in programming languages. Language design efforts which emphasize verifiability within this narrow framework are, therefore, forced to pay to high a price. We argue in favor of maintaining abstraction capabilities and breaking from the microscopic analysis of current verifiers.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

Introduction

The works of Floyd [Floyd 67, 71] and Hoare[Hoare 69,71,72] have helped establish a framework for formal verification of programs. This framework has unfortunately been construed by some researchers in an overly narrow manner which restricts the verifier to local syntactic analysis. However, since the approach has considerable power some researchers have begun to use it as a design criterion for programming languages [London, et. al. 77][Guttag, 77]. It is claimed that the simplicity of a language's Floyd-Hoare axioms can be used as a measure of that language's perspicuity. To achieve this ease of verification within the syntactically oriented framework of most current verifiers, various program primitives and practices have been disallowed. Arguments against Global variables have been made on these grounds. Most recently, EUCLID [London et. al. 77] has disallowed "aliasing" (sharing between two actual procedure parameters) on the grounds that this produces a simpler procedure call rule.

We will show in this paper that this desire for a simple, syntactically oriented axiomitization is in conflict with other more important goals of language design such as naturalness, ease of data abstraction, and simplicity of specification. We therefore regard the proposal to make verification easier by disallowing programming constructs as ill-advised. Instead we will suggest an approach to program verification which is not confined to local syntactic analysis and which might serve as a basis for verification in languages now considered difficult to verify.

The Problem

While there has been much good work done within the Floyd-Hoare framework, most researchers have avoided programs with side effects on shared data structures. Some work has been done on proving properites of programs which manipulate complex data structures [Burstall, 72], but these have explicitly avoided the issue of sharing. Programs which manipulate arrays, however, have been studied to some degree. It seemed reasonable that the framework established in dealing with arrays ought to be extensible to programs with records and manipulable pointers. Suzuki's thesis [Suzuki 76] was an early attempt to make this extension. He provided an axiomitization for PASCAL, including records and pointers. However, the assignment axiom in Suzuki is incorrect and leads to unsound inferences when dealing with shared structures.

Floyd-Hoare systems are constructed by stating axioms which describe the behavior of each primitive of the programming language. To verify a particular program, assertions are attached to the program describing what conditions are expected to hold on entrance and what conditions are promised to hold on exit. The exit assertion is then passed back over each statement of the program, producing a new assertion which reflects the effects of that program statement. An updated predicate is obtained at the entrance side of the program. This is put into an implication

with the entrance condition. If this implication can be proved, the program is said to be consistent with the specifications. More simply we say that it is correct. If the implication is false then the program and its specifications are inconsistent; debugging of either the program or its specifications is required.

The process of producing updated predicates by use of the language defining axioms is called Verification Condition Generation. (Some systems generate the verification conditions in a forward direction. In this paper we will always assume backwards generation. The problems we will show occur in either method.) In dealing with simple programs without sharing it was found that the verfication condition generator needs no information other than the current predicate and the axiom defining the current program statement. Many researchers were led to believe that local syntactic analysis was both desirable and adequate for more complex situations. We will show that there are cases where this approach is overly microscopic, leading to incorrect verification conditions. One consequence of this is that incorrect programs will sometimes be judged to be correct and vice versa.

Consider the following fragment of a PASCAL program which manipulates association lists (association lists are lists of pairs and are used frequently in LISP):

```
type alist = record first: ↑pair; rest: ↑alist;  end;
     pair = record left: integer; right: integer end;

var The_Alist: alist;
    The_pointer : ↑pair;
      .
      .
      .
    The_pointer := The_Alist.first;
    The_pointer↑.left := 3;
      .
      .
      .
end
```

LISP programmers will recognize that this is equivalent to the following:

```
(rplaca (car The_Alist) 3)
```

Let us define some terminology which will make the discussion easier. Members of the alist are the pairs pointed to by the sucessive first pointers of the alist record structure. The left part of a pair is its Key. Suppose that for some reason we were interested in whether the keys of the members of the association list are all even integers (in practice we might be interested in axiomitization for PASCAL, including records and pointers. However, the assignment axiom in Suzuki is incorrect and leads to unsound inferences when dealing with shared structures.

Floyd-Hoare systems are constructed by stating axioms which describe the behavior of each primitive of the programming language. To verify a particular program, assertions are attached to the program describing what conditions are expected to hold on entrance and what conditions are promised to hold on exit. The exit assertion is then passed back over each statement of the program, producing a new assertion which reflects the effects of that program statement. An updated predicate is obtained at the entrance side of the program. This is put into an implication with the entrance condition. If this implication can be proved, the program is said to be consistent with the specifications. More simply we say that it is correct. If the implication is false then the program and its specifications are inconsistent; debugging of either the program or its specifications is required.

The process of producing updated predicates by use of the language defining axioms is called Verification Condition Generation. (Some systems generate the verification conditions in a forward direction. In this paper we will always assume backwards generation. The problems we will show occur in either method.) In dealing with simple programs without sharing it was found that the verfication condition generator needs no information other than the current predicate and the axiom defining the current program statement. Many researchers were led to believe that local syntactic analysis was both desirable and adequate for more complex situations. We will show that there are cases where this approach is overly microscopic, leading to incorrect verification conditions. One consequence of this is that incorrect programs will sometimes be judged to be correct and vice versa.

Consider the following fragment of a PASCAL program which manipulates association lists (association lists are lists of pairs and are used frequently in LISP):

```
type alist = record first: ↑pair; rest: ↑alist;  end;
     pair = record left: integer; right: integer end;

var The_Alist: alist;
    The_pointer : ↑pair;
      .
      .
      .
    The_pointer := The_Alist.first;
    The_pointer↑.left := 3;
      .
      .
      .
end
```

LISP programmers will recognize that this is equivalent to the following:

```
(rplaca (car The_Alist) 3)
```

Let us define some terminology which will make the discussion easier. Members of the alist are the pairs pointed to by the sucessive first pointers of the alist record structure. The left part of a pair is its Key. Suppose that for some reason we were interested in whether the keys of the members of the association list are all even integers (in practice we might be interested in more useful predicates, but the argument we are about to make will hold for these more complex predicates as well). We could express this with a predicate such as:

```
(Even_Keyed  The_Alist)
```

It is obvious that in the above program fragment this predicate is false after the second assignment statement, but that it might be true on entrance to the fragment of code. An accurate verification condition generator, therefore, should modify the predicate in moving it backwards over this assignment statement. Suzuki attempted to state an axiom for assignment which would have this property. The following discussion will show that this attempt failed.

The methodology adopted by Suzuki was a modest extension of the axioms for array assignment. He attempted to make other complex structures resemble arrays by allowing the "array index" to be an object of type other than integers. Thus, a record can be regard as an array in which the "index" is allowed to be a field selector name. LIST_1.REST, for example, can be regarded as the object LIST_1 indexed by REST.

Pointers are a little more complicated. There is no problem at all with the pointer variable themselves, these are just simple variables and handled accordingly. Dereferenced pointers, however, are more complex. P↑ (read "P dereferenced") means the item pointed at by the pointer P. Since P is restricted to point only at objects of a particular type, P↑ is essentially selecting out one object from the "array" of all the objects of that type. Thus, dereferenced pointers which point to a common type are viewed as members of an array which is "indexed" by their name. For example, if P is a pointer of type ↑LIST, we can imagine the object P#LIST as the "array" of pointers to lists. P↑ can then be thought of as P#LIST indexed by P.

This construction is formalized by regarding each reference to a complex structure as being composed of an identifier part and a selector part. The following examples show the identifier and selector parts of typical simple expressions:

Class	Example	identifier	selector
Simple Variable	A	A	none
Array Reference	A[I]	A	[I]
Record Selector	A.rest	A	.rest
Derefrenced Ptr	Ptr↑	P#T	⊂Ptr⊃

In the last example T is meant to be (type_of Ptr↑), i.e. the type of objects to which the pointer can point. These examples are extended to more complex cases by obvious recursion rules. Thus, in the example program above, the identifier and selector parts would be:

The_Alist.first	The_Alist	.first
The_Pointer↑.left	P#Pair	⊂The_Pointer⊃.left

A replacement notation is introduced to represent the effect of an assignment:

<IDENTIFIER, SELECTOR, NEW_VALUE>

This represents the value of IDENTIFIER with the element selected by SELECTOR replaced with NEW VALUE. The obvious reduction rule applies:

```
<Identifier, Selector_1, New_Value> Selector_2 =
        if Selector_1 = Selector_2
           then   New_Value
           else   Identifier⊗Selector_2
```

where the ⊗ denotes concatenation.
Finally, the extended assignment axiom can be stated as follows:

$$P\,|^{A}_{<ari(A),\ ars(A),\ E>} \qquad \{A \leftarrow E\} \qquad P$$

where ARI and ARS are the identifier and selector parts of A, and P is some predicate which is known to hold after the assignment. The expression on the left represents the predicate formed from P by replacing every free occurence of A in P by the expression <ari(A), ars(A), E>. The assignment axiom says that if P is known to hold after the assignment, then the substituted P can be inferred to hold before the assignment. It can easily be seen that if we are confined to arrays and simple identifiers this rule degenerates to the normal array and identifier assignment rules.

Let us consider how this rule would interact with the program fragment we showed above and the simple predicate which we agreed would hold after the program fragment's execution. Here we include the assertions with the code.

```
ASSERT ENTRANCE (Even-Keyed The_Alist);

The_pointer := The_Alist.first;
The_pointer↑.left := 3;

ASSERT EXIT ¬(Even-Keyed The_Alist);
```

A quick examination of the assignment rule will indicate that the exit predicate will be passed over the two statements unmodified (since The_Alist is never mentioned on the left side of the := in either statement). The following verification condition is formed:

(Even-Keyed The_Alist) → ¬(Even-Keyed The-Alist)

Which simplifies to FALSE, signifying that the program is not consistent with the specifications. Now consider what would happen if we had mistakenly written the exit assertion as follows:

```
ASSERT ENTRANCE (Even-Keyed The_Alist);

The_pointer := The_Alist.first;
The_pointer↑.left := 3;

ASSERT EXIT (Even-Keyed The_Alist);
```

In this case it is obvious that the code and the specifications are inconsistent. However, the assignment axiom would lead the verification condition generator to pass the assertion back unchanged, obtaining the implication

(Even_Keyed The_Alist) → (Even_Keyed The_Alist)

which simplifies to TRUE, incorrectly signifying that the code is consistent with the specifications. We, thus, have a case where an overly local and syntactic verifier will tell us that an incorrect program is correct and that a correct one is incorrect.

What Went Wrong?

The crucial fallacy underlying the above failures of the verifier is the assumption that the effect of a program primitive can be determined through microscopic rather than macroscopic analysis. Suzuki states: "The implication of this (assignment) rule is that the meaning of the assignment statements can be determined locally." [Suzuki 76 p. 66]. The assignment axiom assumes that an assignment to an object can effect only that object and therefore that a syntactic substitution rule can be applied without macroscopic reasoning about the extent of the effects.

However, the above example violates these underlying assumptions by using pointers to maintain non-local connections between data objects. Side effects to the pair pointed at by the pointer cause derived side effects in the alist of which it is a conceptual part. This is true even though the two objects (the alist and the pair) are of different types. Moreover, suppose that the alist itself is a sub-part of some larger structure such as a hash table. Then side effects to the pair could result in derived side effects in the hash table. In other words, side effects are an inherently non-local phenomenon requiring macroscopic analysis to determine the scope of the derived effects.

Another way of looking at this is to notice that in the definition of the predicate EVEN-KEYED there is a reference to "free" variables. In order to define a member of the alist we must talk about the pair pointed to by the FIRST pointer of the alist (and recursively for the alist pointed to by the REST pointer). However, there is no mention of these details in the predicate EVEN-KEYED. It is this which accounts for the non-local nature of assignment. Indeed, a possible solution to the problem is to say that such predicates are not allowable, forcing the user to refer more explicitly to the pointers and cells involved.

It is interesting to note that PASCAL (an early structured language) makes all pointers explicit in an attempt to be perspicuous. In contrast, LISP makes all pointers implicit gaining in abstraction power. In LISP one thinks of objects simply as being parts of larger structures, the pointers are part of the implementation and are irrelevant to the programmer. In PASCAL one must worry about the pointers. Ironically, the loss of abstraction in PASCAL does not result in code which is easier for a syntactically oriented verifier to understand. Indeed, we have just seen a simple PASCAL program which is misunderstood by such a verifier.

The reasoning needed to understand this program is the same whether the program is coded in LISP or PASCAL. In either language we need to know that the pair is a part of the alist and therefore that side effects to it might cause derived side effects to its parent structure. Understanding programs necessarily involves reasoning about the connectivity relations between objects. Such reasoning is different than the microscopic view assumed by Suzuki's and similar

systems.

The failures of such overly syntactic verifiers are caused by the violation of the "unique naming property" which states that any object is accessible by exactly one name. When Suzuki states that the "meaning of an assignment can be determined locally," he assumes that the language satisfies this "unique naming property". Pointers, however, violate this property by allowing an object to have one "name" for each pointer which points at it. This undermines the assumptions which underlie the verifier. We will now go on to show other ways in which the assumption can be violated.

Pointers Are Not The Real Problem

In what has become somewhat of a tradition in language design one might try to disallow pointers altogether. The tendency to disallow constructs as a solution to the weaknesses of syntactically oriented verifiers is now well established. Some have argued that global variables should be proscribed for this reason. Indeed, the suggestion that pointers be disallowed has been made seriously in [Ligler 76]. Before we disallow pointers as well, we should consider whether that change alone will be adequate.

For the moment, let us consider a language without pointers but with arrays and integers. Most of us would consider these essential features of any language. We are going to show that these will be sufficient to construct the equivalent of pointers.

Suppose that we have two arrays of integers. The first of these will be used to hold integer "values", the second will be used to index itself. A zero index in this last array will be taken as a "stop" code.

Given an integer one can obtain a series of indices running through the second array by repeated indexing. The reader might have already realized that the two arrays might well be named FIRST and REST since we are using the strings of integers to implement a list structure. (Indeed, these constructs are a simplification of LISP's CAR, and CDR). The REST array contains the "threads" of the list-structure, FIRST contains integer values which are "members" of the list.

We can, therefore, define the relations FIRST_VALUE, SUB_LIST, and MEMBER between two integers as follows:

```
(FIRST_VALUE Integer_1 Integer_2) ≡ (= Integer_2 First[Integer_1])

(SUB_LIST Integer_1 Integer_2) ≡ (= Integer_2 Rest[Integer_1])

(MEMBER integer_1 integer_2) ≡ (Or (FIRST-VALUE integer_1 integer_2)
                                   (MEMBER [SUB_LIST integer_1] integer_2))
```

where the [...] notation is an abbreviation indicating the unique integer which is the sublist of integer_1.

Thus to see if one integer value is a member of the list indicated by a second integer one simply enumerates the chain of indices in the SUB_LIST array which begins at the place indexed by the second integer. For each such enumerated index, the integer in the same position of the FIRST array is tested for equality to the integer value input to the member test. If the two integers are equal, the membership relation holds; otherwise the search is continued until a zero index is found.

Since we have now created list structure of the form used in LISP it should be clear that the arrays and integer indices are being used in a manner behaviorally equivalent to pointers. We will now show that the syntactically oriented verifier which failed to handle pointers correctly will also fail to handle assignments to the above arrays.

Consider the following code fragment:

```
var First, Rest  : Array [1 .. n] of Integer;
X,Y,Z : Integer;
 .
 .
 .
 Rest[Z] := 4;
 .
 .
 .
 ASSERT Exit (Member X Y)
end;
```

This code fragment is subject to the same bug as was the pointer example above. Since the list X might thread through position Z, it is possible that the assignment might splice new members into the list or that it might splice members out of the list. However, the verification condition generator will pass the predicate back intact since neither X nor Y is mentioned in the assignment statment. Once again the syntactically oriented verifier will ignore the possibility of derived side-effects. The fact that Y is a member of the list X after the assignment statement in no way implies that the same was true beforehand. Yet the verifier acts as if it does. Thus, disallowing pointers is an inadequate tactic since we have produced a derived side effect even though there are no pointers at all in our language.

In this case, the verifier was confused by our coding of implicit pointers into the control flow of the program. The integers became pointers by virtue of the way they were used. As long as a language has arrays and integers it will be possible to construct such a scheme. Disallowing pointers will not be sufficient to enforce the unique naming property. It should be noted that in this example as in the previous one, free variables were used in the definitions of the predicates. The membership predicate makes free reference to the two arrays FIRST and REST. We will return to this issue in a moment.

Faced with the problem of our array example above, a truly dedicated devotee of disallowing language constructs might now move on to disallow arrays as well. However, we should take note that it is possible to obtain the behavior of arrays using nothing but integer variables. The bits of an integer can be broken up into sub-strings, so that an integer of N bits can encode N/M integers of M bits. Thus, two integer variables could encode the arrays FIRST and REST using only arithmetic operations. *Integers alone are sufficient to produce derived side effects.* We can safely assume that none of us would argue for measures as draconian as removing integers from our language.

Disallowing constructs, therefore, leads us down a path which terminates in a highly perspicuous but essentially empty language. Nothing can be written in it, but it expresses everything it can in a highly transparent manor. Continuing in this direction is, therefore, untenable; other solutions should be sought.

Other Ways Out

The discussion above has very serious implications. What we have seen is that the use of naturally oriented predicates in our program description language comes into conflict with the

desire to have a simple syntactically oriented verifier. There are now researchers who suggest attacking this conflict by restricting the programming language [Guttag 77][Ligler 76]. However, we have shown that as long as we allow predicates whose definitions have free variables there is no way to restrict programming languages so as to make a local syntactically oriented verifier work as we would desire. One might, therefore, either try to patch the current verifier logic or to develop a different logic for program understanding.

Let us first try to find an incremental patch. As we have mentioned the problem can be characterized syntactically by the presence of free variables in the definition of predicates in the description language. We could disallow this, requiring a predicate of the meta-language to mention explictly everything which is ultimately involved in its truth. (Notice that this is not the same as disallowing global variables in the programming language. We reagrd the program descripton language as a separate language). In our previous example we would then have had to include in all MEMBER predicates the arrays FIRST and REST.

```
(Member Integer_1 Integer_2 First Rest)
```

Side effects to either of the arrays would then be seen as having an effect on membership. The assignment axiom will replace the appropriate symbol (FIRST or REST) by an update expression during verification condition generation. Similarly in the pointer example we would have had to include in the membership predicate for ALIST a representation of pointers to lists and pointers to pairs. These would then also have had to been included in the EVEN-KEYED predicate

```
(EVEN-KEYED The_Alist P#Alist P#Pair)
```

However, we consider this solution unsatisfactory. Our first objection is on practical grounds. For the small list system that we created above, the predicates of the meta-language are already large. In truly large and inter-related systems it would become an unbearable chore to write predicates which syntactically represented all possible dependencies. Each predicate would have to mention nearly everything. As we saw earlier, the failure to include everything could lead to incorrectly deciding that an incorrect program is correct. Furthermore, a verification system which insisted on such excessive detail would simply not be used.

There is a second and more profound objection. We are being forced to include in the predicates of our description language references to the concrete representation underlying an abstract data type. *We are being forced to chose between locality of effect and abstraction of description.* In the array-based list system above what one is interested in is the abstract notions of

being the first item in a list, being a member of the list, etc. At some level, these abstract concepts ought to be independent of implementation issues such as what arrays are used to implement the list system. Nevertheless, the local and syntactic character of our verifier is forcing us to deal with these concrete details at all levels.

Language designers should take note of the tradeoff which this suggests. One cannot design a language which satisfies the locality constraints and syntactic orientation of current verification systems while effectively capturing data abstractions. This tension has expressed itself in the different strategies taken in various language design projects. Languages such as CLU [Liskov 74][Liskov et. al. 77][Liskov & Zilles 74,75] have chosen to concentrate on effective mechanisms for expressing programs abstractly. In contrast, EUCLID [Lampson 77][London et. al. 77][Guttag et. al. 77] was designed with the goal of being verifiable by a verifier such as Suzuki's. Language constructs were restricted intentionally to make verification more tractable. ALPHARD [Wulf 74] seems to have had both goals in mind, providing data abstacting mechanisms, but relying on a syntactically oriented methodology for verification.

Faced with such a choice we would argue for language designs which emphasized abstraction capabilites. Indeed, since abstraction is a fundamental human technique for handling complexity, languages which provide abstracting facilities are crucial steps forward. However, we will show that even with abstraction mechanisms a language can not prevent derived side effects. Therefore, we will argue for a different type of programming logic which allows both abstraction and a correct understanding of the program.

Can Abstraction Mechanisms Fix The Problem?

The clusters of CLU, the forms of ALPHARD and related notions in other languages provide abstraction facilities by defining a data type in terms of functions which represent the possible operations on objects of that type. These functions have common access to the concrete representation, but code outside the abstraction does not. The transition between concrete and abstract representations are intended to take place exclusively within the clusters. Within the cluster, all predicates which depend on the representation should mention every concrete object on which they depend. Outside the cluster, abstract predicates which ignore the representation can be used. This suggests a style of programming in which many layers of clusters are built up, each adding a bit more abstraction. One would hope that this would strike a compromise which might solve the problems we have raised above, allowing data abstraction while preserving the verifier's local and syntactic framework.

However, there are shortcomings in this approach. One is that abstraction mechanism such as CLU's clusters cannot prevent the programmer from making the concrete representation underlying an abstraction available to programs outside the cluster of abstracting functions. (In any event, we do not believe that it is always possible or desirable to program in a strictly hierarchical manner in which each cluster keeps its internals hidden. There is too much need for interaction). However if the representation is exposed, side effects to the concrete objects can cause derived side effects which an overly syntactic system will miss.

A second shortcoming is that in a main routine a programmer might well use encoding devices (such as we used in the array example above) without building up a cluster. Thus, although abstraction oriented languages provide a notion of good coding practice, there is no way of forcing the programmer to respect these notions. Bugs similar to the one in the pointer example are likely to occur.

The Frame Problem

Even if clustering principles are respected the problem of derived side-effects will still remain. Consider an abstraction which implements LISP style lists. This would have 5 operations CAR, CDR, CONS, RPLACA, & RPLACD sharing access to some internal representation. (In what follows we will not know what the representation is nor will we care). How would we specify the behavior of these routines? We would like to say what is obvious, that CONS creates a brand new pair, CAR returns the left half and CDR the right half of the pair. RPLACA and RPLACD change the left and right half of a pair respectively.

However, these specifications are inadequate within the verification framework we have considered so far. If this were all we said, we would run into the problem that predicates such as MEMBER, LENGTH, ORDERED, EQUAL, etc. would all be incorrectly handled. Consider the following:

```
Assert Entrance (LENGTH LIST_1 15)
        .
        .
        .
      (RPLACD   LIST_2  LIST_3)
        .
        .
        .
Assert Exit (LENGTH LIST_1 21)
```

In line with previous examples we can see that a syntactically oriented system which used the specification for RPLACD suggested above might conclude that the code and the specifications were consistent only if 15 = 21. Since the specifications for RPLACD only say that it changes the CDR of its first argument, the LENGTH predicate would be passed over the procedure call unchanged. Notice that we have not exposed the rep and that we are only using operations from the LIST cluster.

To fix this while still maintaining the framework we have used, we would either have to disallow the abstract LENGTH predicate or conclude that the specifications of RPLACD are wrong. Disallowing the LENGTH predicate would negate the entire point of having the abstraction, namely to talk about the behaviour of objects in intuitive terms. We might be willing to be more concrete inside the abstraction mechanism, talking about length in terms of the representaton; outside, however, a more abstract statement is desirable. We, therefore, have to redefine the specifications of RPLACD.

But what would we redefine its specifications to be? To make LENGTH a usable predicate we would have to say that RPLACD can change the length of any list of which the effected list is a sub-list. Similarly, to make the membership predicate usable we would have to include specifics about how RPLACD effected it. Similarly for ORDERED and any other predicates we might want to have on lists.

The specifications for each operation, therefore, would have to tell us exactly how that operation affects every predicate which might eventually be used. Any time a new predicate is added to the system, we would have to augment each operation's specifications with statements describing how it did or did not affect the new predicate. Each of these specifications would then have to be re-verified. The specifications for each function would be long and complicated.

This forces us to violate our intuitive sense of what an operation's specification ought to say. Specifications should state the intrinsic behavior of an operation, i.e. that behavior of an operation which is true no matter what context the operation is employed in. In contrast, an extrinsic description would give a sense of the purpose of a particular instance of the operation. Although the borderline between intrinsic and extrinsic is quite thin and elusive, the specifications above have too much of an extrinsic flavor. Simpler and more clearly intrinsic specifications would be preferable.

This problem also appears in common sense reasoning. For example, suppose I were to say that the box is in the pile in the corner. If I then added that John had thrown out the pile in the corner, most of us would assume that the box was gone. However, if I said that I had painted the pile in the corner, then we would assume that the location of the box had not changed. Somehow our knowledge of "painting" and "throwing things out" tells us that the one affects position while the other does not. Furthermore, our knowledge of what is meant by a "pile" tells us that since the box was in the pile, it was affected by both actions on the pile. This is directly analogous to determining the scope of a derived side effect in programming.

The general problem of telling when an operation will affect a predicate has been called the Frame Problem by McCarthy and Hayes [McCarthy and Hayes, 1967] and has been the subject of considerable Artificial Intelligence research. The Frame Problem can be characterized as follows: If one knows about N operations and M predicates, then one must provide MxN specifications of how each operation affects each predicate. In addition, one is required to do this in a way which is computationally practical. If M and N are reasonably large numbers (as would be the case in common sense reasoning and large programming systems) then the specifications of the NxM interactions would be intractably large. One would be quite likely to overlook some interaction.

A second computational problem is that after each action the verifier (or problem solver in the more general context) must rebuild its entire view of the world. In a description of a program of moderate complexity, this would require the verifier to examine hundreds of predicates each time a program side-effect occurs. If a heuristic is used to reduce this computational burden, it must be guaranteed not to introduce inconsistencies. We have seen that verification systems now in use do introduce inconsistencies by failing to take account of derived side-effects.

Several different approaches to the frame problem have been suggested, however, none of them can claim to be a total solution. Raphael [Raphael 1970] and Hayes [Hayes 1971b] have catalogued many of these. One rather obvious technique is to specify for each operation only the things which it does change, assuming that predicates not mentioned in an operation's specifications are not affected by that operation. This allows a context representation of time in which predicates are automatically carried from one situation into its time sucessor unless the specifications of the operation affecting the transition contain an explicit instruction not to do so. The Artificial Intelligence programming languages QA4 [Rulifson, et. al. 1972] and CONNIVER [McDermott & Sussman, 1972] were designed with such features.

A second technique is to classify the predicates into a frame [McCarthy & Hayes 1967] which divides them into blocks of non interacting types. For example, color and position would be in different blocks of the frame. Notice that this is a predicate oriented classification which is distinct from the object oriented classification of data abstraction languages.

Another method was added in the STRIPS system [Fikes & Nilsson 1971]. Predicates were classified as primitive or derived. An operation was required to specify only how it effected the primitive predicates, its effect on a compound predicate was deduced. PLANNER [Hewitt, 1972] and its descendent languages added the ability to provide specifications in the form of procedures which are triggered whenever a matching predicate is added or erased. In PLANNER one can code a simple procedure which states that if an object is moved then the position of all objects on top of the moved object should also be changed. For historical reasons these procedures are called erasing theorems.

Hayes [Hayes 1971a] has suggested that a representation of the causal relationships between facts be maintained in the reasoning system. This allows the system to focus its attention. If a fact is changed, then the system need only examine facts which are causally related to the modified fact.

It comes as a surprise to many people that the frame problem is a problem at all. People seem to handle it so easily. However, a mechanical reasoning system can act as effectively as people do only if it has built into it techniques and knowledge of comparable power to those which people use. The syntatically oriented verification systems we have examined earlier do not in general have such powerful techniques. In fact, Suzuki's verifier and its predecessors use a very simple and weak strategy. These systems assume that each operation changes properties of only those objects mentioned explictly in the operation. Everything else is assumed to remain as it is. However, we have seen that this assumption can be maintained only be making unreasonable demands on the descriptive language.

A Different Approach to Verification

Language designs which attempt to enforce good coding practices are limited in what they can accomplish. One can hope for a language which encourages a perspicuous programming style, but there is no way to enforce a style which will allow a verifier to use microscopic analysis exclusively. Languages such as EUCLID which do attempt to enforce such restrictions will be cumbersome.

Verification systems which depend on the programmer's adherence to "good coding practices" are likely to become too brittle. In some cases they will be able to do what is needed, in others they will be able to provide no help at all. What is desired is a system which is more flexible. Program verification systems should be able to function with a partial understanding of the program, providing useful assistance for the part they do understand and requesting more information for the parts which they do not. Such a system would not depend on the existence of a well structured programming language, although the existence of one might help it do its job.

A system of this type is somewhat analogous to a junior programmer. It has a substantial body of programming knowledge to draw on, but it doesn't know everything. The senior programmer describes a program to the partner in high level terms; the junior partner builds an internal model of this description which links together the dependencies and interrelationships which he understands. This junior partner can then be assigned routine tasks such as checking the correctness of a module since it knows enough about the system to do this effectively. Because of the analogy, systems of this type have been termed programmer's aprentices [Smith & Hewitt, 75][Rich & Shrobe, 76].

In an apprentice system, there is no demand for locality of program primitive effect. Thus, in a LISP apprentice one might state that the effect of RPLACA (replacing the first of a list) is simply to change the first item of the list. However, we would leave it to the apprentice to discover the derived side effects. There would be no need to enumerate these in the description of RPLACA. Typically, the apprentice system would already know the meaning of concepts like MEMBER, LENGTH, SORTED, etc. If it didn't, it would ask the programmer to define these predicates. These are then kept in a knowledge base of programming concepts in a form suitable for associative retrieval. For example, the programmer might tell the system what he means by membership in a list as follows:

```
(Relation-Definition
 (Member list object) ≡(or (first list object)
                           (member [rest list] object)))
```

where the brackets are an abbreviation meaning the unique object which fits the description given in the brackets. This definition is inverted to discover potentials for derived side effects. This information is also added to the knowledge base.

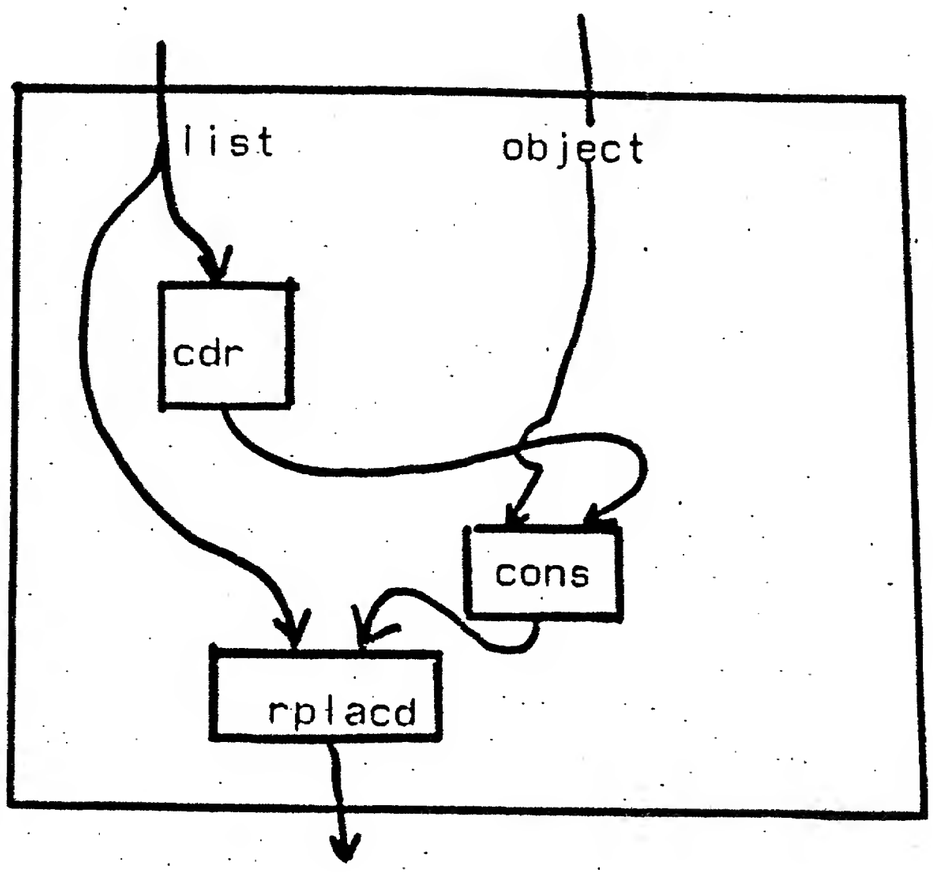
```
(Depends-on ((first list object))
            (member list object))

(Depends-on ((member list-2 object)
             (rest list-1 list-2))
            (member list-1 object))
```

Where in the second form the subscripts have been inserted by the system to distinguish between the list and its immediate sublist. To see how the system would make use of these descriptions let us look at how it would analyze a a list insertion routine such as the following:

```
(define list-insert (list object)
        (rplacd list (cons object (cdr list))))
```

Our system would first translate this into an internal flow diagram as follows.



This diagram would then be symbolically evaluated in a situational data base which represents the intermediate states of the computation. As each module is evaluated we prove that its preconditions can be derived from the facts present in its situation of application. If this proof is sucessful we create a new output situation and assert in this those facts which result from the execution of the module. If a side effect results we use the dependency information in the data base to deduce the derived side effects. In the above program we would obtain the following situation map:

Situation-0

```
(list list-1)
(object object-1)
```

Situation-1

```
(list list-2)
(cdr list-1 list-2)
```

Situation-2

```
(list list-3)
(car list-3 object-1)
(cdr list-3 list-2)
(member list-3 object-1)
```

Notice that the relation definition of membership was used in situation-2 to infer that object-1 is a member of the newly created list list-3. The rplacd is now applied in situaton 2. Given that its specification is simply that it changes the cdr we obtain the following situation:

Situation-2	Situation-3
(list list-1)	
(object object-1)	
(list list-2)	
(cdr list-1 list-2)	(not (cdr list-1 list-2))
(list list-3)	
(car list-3 object-1)	
(cdr list-3 list-2)	
(member list-3 object-1)	
	(cdr list-1 list-3)
	(member list-1 object-1)

Notice that the assertion (MEMBER LIST-1 OBJECT-1) is a derived side effect which is deduced by using the dependency information in the knowledge base.

This approach to the problem of derived side effects draws on several of the techniques for handling the frame problem. The examination of the predicates for dependency information is similar to Hayes's notion of maintaining causality relations. The propagation of the derived side effect is handled in a manner similar to that of PLANNER's erasing theorems. Finally, the specification of operations has the flavor of STRIPS's division into primitive and derived predicates.

This has been only a cursory introduction to the kind of reasoning which the system must engage in to maintain an accurate symbolic representation of the program's dynamic behavior. In fact, given an arbitrary program containing side effects on complex objects, it might be impossible to deduce whether a particular predicate continues to be true after the side effect has happened. For example, suppose one of two lists is subjected to a RPLACD. In the absence of other information, we would not know whether the second list was also effected. However, if we knew that the lists shared no structure then we would know that the second list was unaffected.

Our system is conservative, deducing what it might not be able to know and maintaining a representation which is always consistent and as complete as possible. In the course of the symbolic evaluation of the program, the system creates processes called constraint propagators which are triggered by pattern directed invocation and which move information between situations. Constraint propagators represent the effect of a module's execution. They can move information

both forwards and backwards in time; however, they are conservative, only moving information if they can prove that it is appropriate to do so. The justification for every deduction is recorded in a permanent data base, allowing the system to use advanced hypothetical reasoning techniques such as truth maintainence, and dependency directed backtracking [Doyle 1977]. The programmer's apprentice system has been described in more detail in [Rich & Shrobe 1976].

Notice that in designing such a reasoning system we have fundamentally broken from the microscopic character of current automated Floyd-Hoare verification systems. We have chosen to pay the price macroscopic reasoning after each program step. In return, we can allow the programmer the flexibility of structuring his code in whatever manner he finds comprehensible and of describing the program in abstract terms without having to include awkward specifications whose sole function is to make the verifier work. The system is quite capable of dealing with programs which have side effects on complex and shared data structures and we believe that the dependency oriented architechture will allow it to be used as part of a flexible design tool.

Conclusions

Floyd-Hoare logic is an attempt to assign a meaning to programs. One practical goal which underlies this attempt is the construction of mechanical verifiers which can understand humanly generated programs while yet being free of the bugs which seems intrinsic to humans. Unfortunately most verification systems which have been developed tend to interpret the Floyd-Hoare methodology in an overly local and syntactic way. As we have seen, this microscopic approach contradicts the need to discuss side effects on shared structures in abstract terms.

Two solutions to this contradiction are possible. The first would restrict programming languages in an attempt to make the verifier work while still maintaining its microscopic approach. There are principled limitations to this approach which suggest that it is simply not tenable. However, if languages are not so constrained, current verifiers which use a local and syntactically oriented methodology will encounter situations in which they misconstrue the program.

While various less drastic changes to the verification methodology are possible, we feel that most of these are too cumbersome and contradict our desire for natural and abstraction oriented programming and description languages. We believe it to be a more reasonable approach to develop a much more powerful logic of programs which reasons macroscopically about program behavior. Using such a logic system there would be neither reason nor need for the current trend of restricting programming constructs.

Acknowledgements

I would like to thank Gerry Sussman, Carl Hewitt, Joel Moses and Guy Steele for encouraging me to put these ideas down on paper at long last. Chuck Rich, Mark Miller and Mitch Marcus were extremely helpful and supportive. Vaughan Pratt's work on semantics (which was the topic of my area exam) was the first place where I noticed some of these problems. Finally, thanks to Craig Schaeffert, Irene Grief and Valdis Berzins for an electrifying argument.

Bibliography

Burtall, R.M., 1972 "Some Techniques for Proving Properties of Programs Which Alter Data Structures", Machine Intelligence 7, Edinburgh University Press.

Deutsch, L.P. 1973, An Interactive Program Verifier, PhD. Thesis University of California at Berkeley, June 1973.

Doyle, Jon 1977 Truth Maintenance Systems for Problem Solving, M.I.T. Artificial Intelligence Laboratory M.S. Thesis May 1977. Also to appear as M.I.T. Artificial Intelligence Laboratory Technical Report 419, 1977.

Fikes, R. and Nilsson, N. 1971, "STRIPS: A New Approach to the Application of Theorm-Proving to Problem-Solving" Proceedings of the 2nd International Joint Conference on Artificial Intelligence, London.

Floyd, R. W. 1967 "Assigning Meaning to Programs", Mathematical Aspects of Computer Science. J.T. Schwartz (ed.) vol. 19, Am. Math. Soc. pp. 19-32. Providence Rhode Island.

Floyd, R.W. 1971 "Toward Interactive Design of Correct Programs", IFIP, 1971.

Guttag, John; Horning, James J.; & London, Ralph L. 1977, "A Proof Rule for Euclid Procedures", Proceedings of The IFIP Working Conference on Formal Description of Programming Concepts, August 1977.

Hayes, Patrick 1971a, "A Logic of Actions", Machine Intelligence 6, Meltzer and Michie eds. Edinburgh University Press, pp. 495-520.

Hayes, Patrick 1971b, "The Frame Problem and Related Problems in Artificial Intelligence" Stanford University Artifical Intelligence Memo 153.

Hewitt, Carl 1972, Description and Theoretical Analysis (using Schemata) of PLANNER, a Language for Proving Theorems and Manipulating Models in a Robot, M.I.T. Artificial Intelligence Laboratory Technical Report TR-258, April 1972.

Hoare, C.A.R 1969, "An Axiomatic Basis for Computer Programming", Comm. ACM, vol. 12, number 10, October 1969, pp. 576-580,583.

Hoare, C.A.R 1971, "Proof of A Program: Find", Comm. ACM, vol. 14, number 1, January 1971, pp. 39-45.

Hoare, C.A.R. 1972 "Proof of Correctness of Data Representations", Acta Informatica, 1,4, pp. 271-281.

Hoare, C.A.R. and Wirth, N. 1973 "An Axiomatic Definition of the Programming Language PASCAL", Acta Informatica, 2,4, pp. 335-355.

Igarashi S., London R., and Luckham D. 1973, Automatic Program Verification I: A Logical Basis and Its Implementation, Stanford AIM-200, May 1973.

King, J. 1969, A Program Verifier, Carnegie Mellon University, 1969.

King, J.C. 1971 "Proving Programs to be Correct", IEEE Trans. on Computers, C-20, 11, Nov. 1971.

Lampson, B.W., Horning, J.J, London, R.L., Mitchell, J.G., & Popek, G.J. 1977, "Report on the Programming Language Euclid", SIGPLAN Notices, Vol. 12, No. 2, Februray 1977.

Ligler, G.T., 1976 "The Assignment Axiom and Programming Language Design", Proceedings of ACM-76 Houston Texas, October 1976.

Liskov, B. 1974, "A Note on CLU", MIT/Computation Structures Group Memo 112, MIT/LCS, November 1974.

Liskov, B.; Snyder, Alan; Atkinson, Russell; and Schaffert, Craig; 1977, "Abstraction Mechanisms in CLU", Communications of the ACM, August 1977, pp. 564 - 576.

Liskov, B. and Zilles S. 1974 "Programming with Abstract Data Types", Proc. of Conf. on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4.

Liskov, B. & Zilles, S.N. 1975, "Specification Techniques for Data Abstractions", IEEE Transactions on Software Engineering, Vol. SE-1 No. 1, March 1975.

London, R.L., Guttag, J.V., Horning, J.J., Lampson, B.W., Mitchell, J.G., & Popek, G.J. 1977, "Proof Rules for the Programming Language Euclid", Technical Report, May 1977.

McCarthy, John & Hayes, Patrick 1969, "Some Philosophical Problems From The Standpoint of Artificial Intelligence", Machine Inteligence 4, B. Meltzer and D. Michie, eds. Edinburgh

University Press, Edinburgh , Scotland, 1969.

McDermott, D.V. and Sussman, G.J. 1972, "The CONNIVER Reference Manual", M.I.T. Artificial Intelligence Laboratory, Memo 259.

Popek, G.J., Horning, J.J., Lampson, B.W., Mitchell, J.G., & London, R.L. 1977, "Notes on The Design of Euclid", Proceedings of the ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, Vol. 12, No. 3, March 1977.

Pratt, V. 1976, "Semantical Considerations on Floyd-Hoare Logic", MIT/LCS/TR-168, September 1976; also in Proc. 17th Ann. IEEE Symp. on Foundations of Comp. Sci., pp. 109-121, 1976.

Raphael, Bertram, 1970 "The Frame Problem in Problem-Solving Systems", Standford Research Institute, Artificial Intelligence Group, Technical Note 33, June 1970.

Rich C. and Shrobe H. 1976, An Initial Report On A LISP Programmer's Apprentice, MIT/AI/TR-354, December 1976.

Rulifson, J.F., Dersken, J.A., and Waldinger, R.J. 1972, "QA4: A Procedural Calculus for Intuitive Reasoning", Stanford Research Institute, Artificial Intelligence Center, Technical Note 73, Menlo Park, California.

Smith, Brian & Hewitt, Carl 1975, "Towards a Programming Apprentice", Proceedings of the IEEE Transactions of Software Engineering, Vol. 1 No. 1, pp. 26 - 45.

Suzuki, N. 1976, Automatic Verification of Programs with Complex Data Structures, Stanford AIM-279, February 1976.

Wulf, W. A. 1974, "ALPHARD: Towards a Language to Support Structured Programming", Carnegie Mellon University, Department of Computer Science, April 1974. Yonezawa, A. 1975, "Meta-Evaluation of Actors With Side Effects", MIT/AI Working Paper 101, June 1975.

Yonezawa, A. 1976a, "Symbolic-Evaluation As An Aid To Program Synthesis", MIT/AI Working Paper 124, April 1976.

Yonezawa, A. 1976b, "Symbolic Evaluation Using Conceptual Representations For Programs With Side-Effects", MIT/AI Memo 399, December 1976.

Yonezawa, A. 1977, "Verification and Specification Techniques for Parallel Programs Based on

Message-Pasing Semantics". M.I.T. PhD. December, 1977